INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 323

MODELLING DATA DEPENDENCIES IN LOGIC PROGRAMS BY ATTRIBUTE SCHEMATA

Pierre DERANSART
Jan MALUSZYNSKI

Juillet 1984

MODELLING DATA DEPENDENCIES IN LOGIC PROGRAMS BY ATTRIBUTE SCHEMATA

Authors :

Pierre DERANSART
INRIA
Domaine de Voluceau - Rocquencourt
BP 105
78153 Le Chesnay Cedex France
Phone : (3) 954 90 20 ext 3536

Jan MALUSZYNSKI
Department of Computer and Information Science
Linköping University
S - 58183 Linköping, SWEDEN
Phone : (13) 1117 001483

Résumé :

Ce rapport fait partie d'une étude visant à analyser les relations qu'il peut y avoir entre les grammaires d'attributs et la programmation logique utilisant des Clauses de Horn. Ces deux formalismes ont été développés indépendemment avec des motivations différentes. Les grammaires d'attributs ont fait l'objet d'études poussées et sont essentiellement utilisées en métacompilation. Elles permettent de décrire le flot de données calculées dans un arbre syntaxique au cours de la compilation. Nous montrons dans ce papier qu'elles peuvent être aussi utilisées pour modèliser le flot de données dans des programmes logiques et nous discutons une stratégie d'évaluation résultant de cette approche.

Abstract :

This report is a part of a study whose objective is to analyse relationships between Attribute Grammars and Horn clause logic. These two formalisms have been independently developed with different motivations. Attribute Grammars have been extensively studied since 1968 and applied mostly in metacompilation.

They make it possible to describe data flow during syntax-directed compilation. In this paper we show that they can be also applied to model data flow in Horn clause logic programs and we discuss an évaluation strategy resulting from this approach.

Mots clé : Programmation Logique, Grammaires d'Attributs, Calculs dirigés par les données, correction de programme.

Key words : Logic Programming, Attribute Grammars, Data Driven Computation, Program Correctness.

1 - Introduction

Better understanding of the information flow during interpretation of logic programs may have a great impact both on the technology of compilation of logic programs [Me 81] , [Ni 83] and on improving efficiency of their interpretation. It may also facilitate the task of organizing parallel interpretation of logic programs.

The paper shows that a proof technique introduced for attribute schemata [De 83] may be useful to deal with some properties of logic programs. To illustrate this we consider a possibility of modelling information dependencies of logic programs by attribute schemata. In particular we deal with the problem wether a logic program called with some arguments instantiated to ground terms returns ground instantiations to the other arguments. This property, when discovered by static analysis, might be used by a compiler to produce more efficient code. It is worth noticing that many examples known from the literature have this property, but for some applications the use of non-ground terms is essential. The paper gives a sufficient condition for a logic program to have the property mentioned . The restricted class of logic programs satisfying this condition does not include well-known examples for which the use of variables is essential. However , every turing machine can be simulated by a program of this class. Data flow of the programs satisfying the condition can be modelled by attribute dependency graphs.

The logic programs dealt with in this paper are finite sets of Horn Clauses. The reader is assumed to be familiar with basic notions concerning the syntax and the semantics of such programs.

Each successful computation of a logic program may be thought of as construction of a complete proof tree. The tree is constructed by using a resolution proof procedure and can be otherwise obtained by pasting together appropriate instances of clauses of the program. Multiple occurrences of variables within a clause result in data dependencies between positions of the labels within the tree. This can be illustrated by the following example.

The clauses :

append ([ ],L,L) ⇐

append ([E|L1], L2, [E|L3]) ⇐ *append (L1,L2,L3).*

may give rise to the proof tree given in Fig 1. The data dependencies defined by the clauses used in this tree are represented by the data dependency graph shown in Fig 2. The nodes of the latter are either positions of the predicates labeling the proof tree or some nodes representing data exchange "actions". These actions correspond to the variables of the clauses occuring in the proof tree since every occurrence of variable within a clause must be replaced in the proof tree by the same term. The graph of Fig.2 can be obtained by pasting together the dependency graphs, which may be drawn for each clause of the program, as illustrated in Fig3. If we assume that the positions of the predicates of the program are classified as inputs we may use this assumption to make the dependency graph of Fig.2 directed. For this we adopt the convention thats inputs have assigned the top-down direction, and outputsthe bottom-up direction. Then using the drawing convention of Fig.2 we assign to every edge of the dependency graph the direction of the position to which it is connected. Thus assignment of the i/o directions to the positions of the predicates of a logic program determines for each proof tree of this program a unique dependency graph spanned on this tree. Assigning to the predicate append the directions ↓↓↑ we obtain for the proof tree of Fig.1 the directed dependency graph of Fig.4.

The question arises, whether the directions assigned by an arbitrary choice to the edges of the dependency graph describe the information flow during the computation. To discuss this question we assume that input data of a logic program are ground terms. The assignment may be considered a model of the information flow if for every procedure of the program called with all its input positions instantiated to ground terms any successful computation returns ground instantiations for all output positions of the procedure. One can easily see that the i/o assignment in the example above has this property. In the sequel we give a sufficient condition for an assignment to have this property. However, for some programs no such assignment may exist.

The dependency graphs for the clauses of a logic program, like those of Fig.3 can be formally specified by means of attribute schemes, introduced in the literature [De 83] in connection with the notion of attribute grammar. The dependency graphs for clauses induce the dependency graphs for proof trees, like those of Fig.3. If the former are specified by means of an attribute scheme then the methods developed for attribute schemes can be used to study properties of the latter.

The rest of the paper is organized as follows. Section 2 gives necessary definitions concerning attribute schemes. Section 3 describes a construction, which for given logic program and direction assignment of its predicate positions results in an attribute scheme specifying formally data dependencies within any proof tree of the program. Section 4 deals with the problem whether a given direction assignment can be considered a basis for a data flow model of a given program. We accept the criterion that a successful call of any procedure with all input arguments being ground should return ground instantiations of all output arguments of the procedure. Section 4 gives a sufficient condition for an assignment and a program which guarantees that the program with this assignment has the property stated above. Section 5 deals with a special class of annotaded logic programs which allow a data driven interpretation strategy.

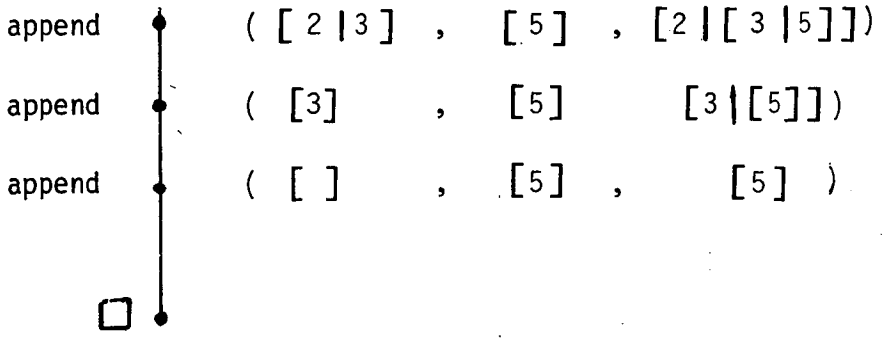


Figure 1. Example proof tree

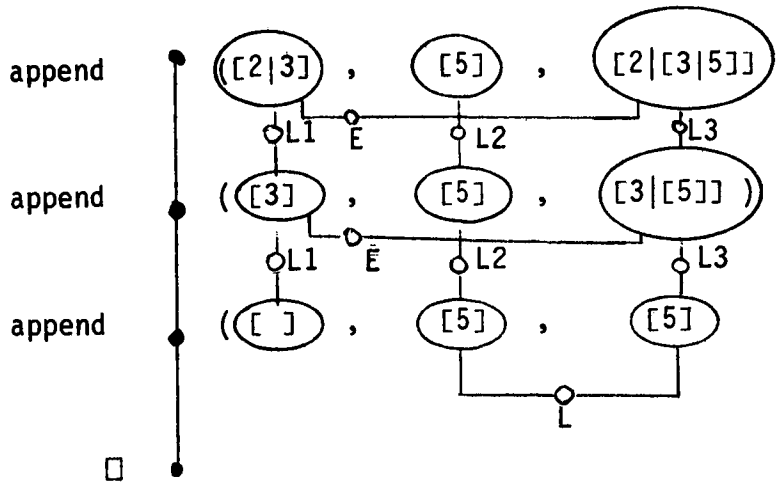


Figure 2. Data dependancies within an example proof tree

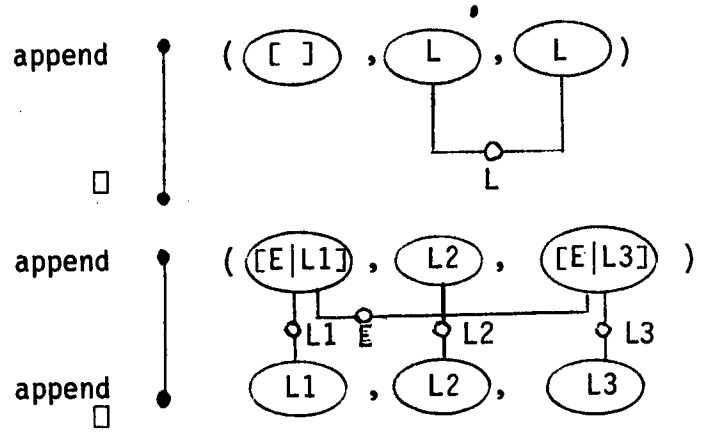
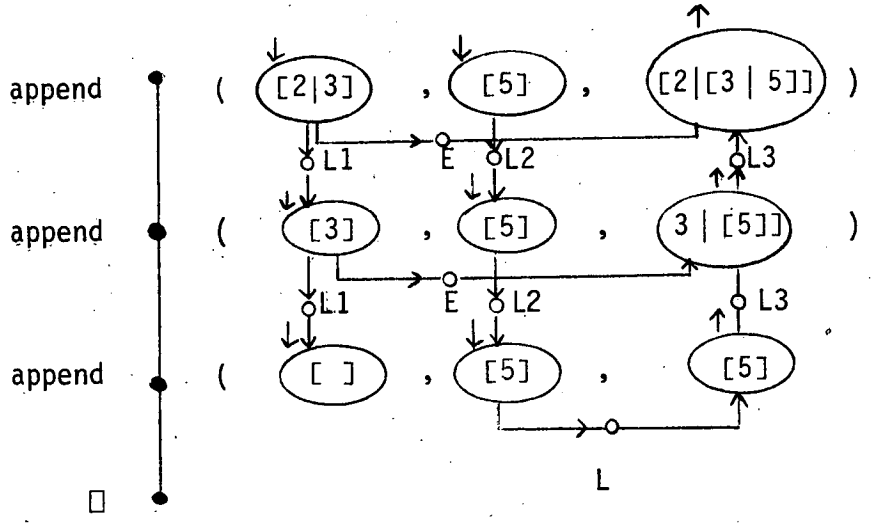


Figure 3. Data dependancies within the clauses of an example



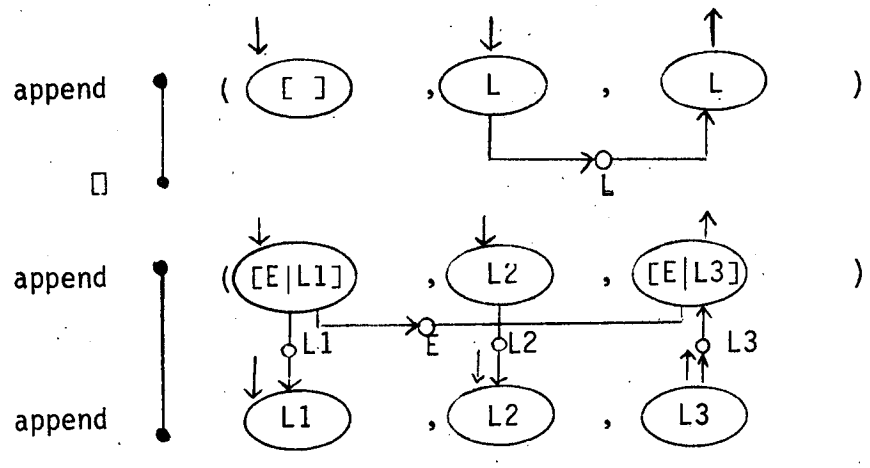


Figure 4. Directed dependency graphs

2 - Attribute Schemes, Basic Definitions

The notion of attribute grammar has been introduced as a specification tool for assigning semantic values to the nodes of parse trees of a context-free grammar. In the same time the formalism makes it possible to compute the values for a given tree, if some additional conditions are fulfilled. Various aspects of the formalism have been extensively studied, like its expressive power, computational strategies, restrictions allowing efficient, computations, complexity of the tests checking whether such restrictions are fulfilled by a given attribute grammar, etc. An extensive bibliography can be found in [Ra 80].

The objective of this paper is to model data dependencies. This means that we are interested rather in possible order of computation of the semantic values than in the values computed. For this purpose it suffices to use only the notion of an attribute scheme, which leaves uninterpreted the operations which are executed to compute the semantic values but describes possible ordering of execution. An attribute scheme can be extented to an attribute grammar by defining interpretation of these operations, called *actions*. This section introduces the notion of attribute scheme as it is used in this paper.

Let G be a context-free grammar

With every nonterminal X of the grammar a number n_X is associated and the semantic value assigned to any derivation tree node labeled X is an n_X-tuple of objects belonging to *some* semantic domains. The positions of the n_X-tuple, also called in the literature attributes of X, are divided into two classes, called *synthesized* (denoted $\uparrow$) and *inherited* (denoted $\downarrow$). Formally, we shall assume that a function d is given, assigning to each X an element of $\{\uparrow,\downarrow\}^{n_X}$. This function will be called a *direction assignment* (shortly d-assignment). By $\downarrow X$, we shall denote the indices of the inherited positions of X. Thus $\downarrow X = \{i \mid d(X)\,[i] = \downarrow\}$ (where the square brackets denote selection of the i-th element of the n_X-tuple d(X)) .

Similarly

$$\uparrow X = \{ i \mid d(X)\,[i] = \uparrow \}.\ (i \geq 0 \text{ in both cases}).$$

The semantic value of a node is defined by relating its components with te components of the semantic values of the neighbouring nodes.

Since the neighbouring nodes are determined by a production rule of the grammar the semantic values are being related at the level of production rules. For this we introduce the following notation :

$$\text{Let } p : X_0 \rightarrow \alpha_0\, X_1 \ldots X_n \alpha_n$$

where X_k are nonterminals α_k strings over the terminal alphabet of the grammar.

$$\text{Let } \downarrow p_k = \{ (k,\ell) \mid \ell \in \downarrow X_k \} \text{ the set of inherited positions of } X_k$$
$$\uparrow p_k = \{ (k,\ell) \mid \ell \in \uparrow X_k \} \text{ the set of synthesized positions of } X_k.$$

Thus every component of the semantic value assigned to an instance of a production rule can be identified by a pair of indices, called a *position* of the rule.

We distinguish between the *input* positions

$$\text{input } (p) = \downarrow p_0 \cup \bigcup_{n>0} \uparrow p_n$$

and the *ouput* positions

$$\text{output } (p) = \uparrow p_0 \cup \bigcup_{n>0} \downarrow p_n$$

DEFINITION 1 : Attribute Scheme

An attribute scheme is a quintuple S = (G, d, A, in, out) where

- $G = (V_N, V_T, P)$ is a set P of context-free production rules with a nonterminal alphabet V_N and a terminal alphabet V_T
- d is a direction assignment on V_N
- A is a finite set of symbols, called *actions* which is the disjoint union of A_p $(p \in P)$.
- in is a function assigning to each $a \in A_p$ a subset of the input positions of p.
- out is a function assigning to each $a \in A_p$ a subset of the output positions of p such that for every p

$$\bigcup_{a \in A_p} \text{out}(a) = \text{output}(p)$$

Attribute scheme formalyses and generalizes in some abstract way the dependencies between attributes existing in attribute grammars. In the case of usual attribute grammar every action will recieve an interpretation, and codomain of out will be a singleton.

Let S = (G, d, A, in, out) be an attribute scheme. For a given production rule p of G,denote by pos(p) the set of all positions of p. Thus :

$$\underline{pos}(p) = \bigcup_i \downarrow p_i \cup \bigcup_i \uparrow p_i,\ 0 \le i \le n_p$$

Thus

$$\underline{pos}(p) = input(p) \cup output(p)$$

A position ν of p *depends on* a position μ of p ($\mu \rightarrow \nu$) iff there exists an action a in A_p such that $\mu \in \underline{in}$ (a) and $\nu \in \underline{out}$ (a). So A, and "out" define a partial ordering between the positions of a production p called the *local dependency relation*, denoted LD(p) (or $\underset{p}{\rightarrow}$)

Derivation trees of G are constructed by "pasting togeter" instances of production rules of G. For a given tree τ, we can enumerate its nodes (e.g. using Dewey indices). We shall assume that such an enumeration is given. By a *position of a tree* τ we mean any pair (n,i) such that n is an index of a node of τ labeled by a nonterminal X and $i \in \downarrow X \cup \uparrow X$.

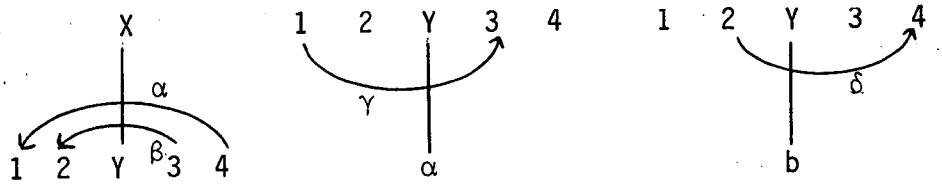
Clearly, given a derivation tree τ of G, the local dependency relations will induce a *dependency relation between positions of this tree*, denoted PD(τ) (or $\underset{\tau}{\rightarrow}$).

An attribute scheme is said to be *well-formed* iff for each derivation tree τ in the underlying CF-grammar the relation PD(τ) is a partial order.

Here we give a classical example of well-formed attribute scheme (Actions can be interpreted as identities)

Example 1 S = (G, d, A, in , out)

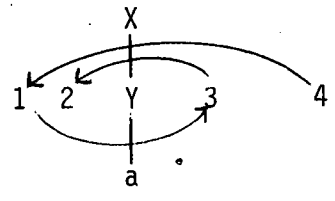
- In G, P = { X → Y, Y → a, Y → b}
- d : X has 0 positions
  Y has 4 positions : {1,2} inherited, {3,4} synthetized
- A = {α,β,γ,δ}, every action has one input one output position.
- in, out are given by the following schemes



It is easy to see that such example satisfies the definition and that the local dependency relation is defined by (the arrows correspond to the local dependency relation) :

$\mu \underset{p}{\rightarrow} \nu \iff \exists a \in A_p$, $\mu \in \underline{in}(a), \nu \in \underline{out}(a)$

It is also easy to verify that in each of the five possible partial trees which can be derived from this grammar the induced dependency relation is a partial order. To illustrate this fact we show here one of the complete trees :



So the partial order (total here) between the positions is

$4 \rightarrow 1 \rightarrow 3 \rightarrow 2$.

Given a subtree τ^X of root X, the induced relation defines a partial order between positions of the roots as follows :

$$\forall \mu, \nu \in \downarrow X \cup \uparrow X,\ \mu < \nu \iff \mu \in \downarrow X,\ \nu \in \uparrow X,\ \mu \underset{\tau^X}{\longrightarrow} \nu$$

We will call use (τ^X, ν) the useful inherited attributes (or useful set) of $\nu \in \uparrow X$ defined as

$$\underline{\text{use}}\ (\tau^X, \nu) = \{\mu \mid \mu \in \downarrow X, \mu \underset{\tau^X}{\longrightarrow} \nu\}$$

Clearly, the useful set of a synthetized position ν of X in the subtree τ^X is the set of inherited positions of X from which ν depends in the partial order $\underset{\tau^X}{\longrightarrow}$.

3 - Associating Attribute Schemes with Horn Clause Programs

Let P be a Horn clause program, P the set of predicates occuring in P, and C the set of clauses of P. For a given direction assignment d on P we construct an attribute schemes S = (G, d, A, in, out) which schould give a formal framework for dealing with dependency graphs like those of Section 1. The data dependencies on proof trees, as those of Fig.3, will be formalized now as the dependency relations on parse trees of the attribute scheme. The underlying context-free grammar is constructed in such a way that the "skeleton" of every proof tree of the program is a parse tree in this grammar.

CONSTRUCTION 1

1. *The underlying CF grammar* G = (N,T,R) *is constructed as follows :*
   N = , T = {□} *(the empty clause)*
   R = $\{r_c \mid c \in C\}$ *where for each clause* c *of the form :*
   $p_0 t\ell_0, p_1 t\ell_1, \ldots, p_n t\ell_n$ *with* $p_i \in P$, $t\ell_i \in term^*$,
   r_c *is the production rule :*
   $p_0 \leftarrow p_1 \cdots p_n$

   Note that R *may include different copies of the same production rule.*

2. d *is the arbitrarily given direction-assignment. The number* n_p *of assigned positions for given* P *is the number of arguments of* p *in the logic program.*
3. *For each* $r_c \in R$, A_{r_c} *includes the set* V_c *of variables occuring in* C. *Moreover, for every position* $\gamma = (i,j)$ *such that* $t\ell_i[j]$ *is a ground term, there is a separate action* a *in* $A_{r_c} - V_c$
4. *For each* $v \in V_c$
   in (v) = {(i,j) | v *occurs in* $t\ell_i[j]$ ∩ input (r_c)
   out (v) = {(i,j) | v *occurs in* $t\ell_i[j]$ ∩ output (r_c)

   For each $a \notin V_c$ *in* A_{r_c}
   in (a) = {p} ∩ input(c) *where* p *is the position corresponding to* a.
   out(a) = {p} ∩ output(c)

One can easily see that the construction reflects the intuitions illustrated in Fig.3: it introduces dependencies between all and only those input positions and output positions of a clause which have a common variable. In the sequel we shall refer to these dependencies when considering instances of clauses. The variables occuring in a clause may not appear in its instance but the attribute scheme still describes dependencies between positions of the instance in terms of the actions introduced in the definition of the attribute scheme.

In the sequel we will sometimes say that a variable has an input (an output) position in a clause iff it occurs in such a position.

To illustrate Construction 1 by an example consider the *append* program and its direction assignment as described in Section 1. The underlying context-free grammar has the following production rules :

(1) *append* $\longrightarrow \varepsilon$

(2) *append* $\rightarrow$ *append*

The actions of the scheme are the following

$A_{(1)} = \{E, a\}$

$A_{(2)} = \{E, L1, L3, L3\}$

The inputs and the outputs of the actions are defined as follows :

$\underline{in}(L) = \{(0,2)\}$, $\underline{out}(L) = \{(0,3)\}$
$\underline{in}(a) = \{(0,1)\}$, $\underline{out}(a) = \{\ \}$
$\underline{in}(E) = \{(0,1)\}$, $\underline{out}(E) = \{(0,3)$
$\underline{in}(L1) = \{(0,1)\}$, $\underline{out}(L1) = \{(1,1)\}$
$\underline{in}(L2) = \{(0,2)\}$, $\underline{out}(L2) = \{(1,2)\}$
$\underline{in}(L3) = \{(1,3)\}$, $\underline{out}(L3) = \{(0,3)\}$

The number of attribute schemes that may be associated with a given logic program by Construction 1 equals the number of different direction assignments. This number may be very large and we would like to focus our attention on some particular assignments. Intuitively, we want the assignments to have the property, that every procedure of the program if called with ground inputs returns ground outputs after any possible successful termination.

More precisely, an assignment is said to be correct for a given program iff for any atomic formula $p(t_1,\ldots,t_n)$ *such that its inherited positions are ground any refutation of the formula determines ground instantiations for all variables occuring in the formula.*

3 - A Sufficient Condition for Correctness of Direction Assignments

Clearly, for some logic programs no correct assignment may exist. However, for numerous programs which can be found in the literature correct assignments can be found, and if given explicitly, make understanding of the programs much easier. In this section we give a sufficient condition for an assignment to be correct for a given program.

We suppose now that d-assignment may be partial, i.e. that it concerns only some positions of the predicates of the program. This makes our results applicable to a larger class of logic programs and covers as a special case total d-assignments. The construction of Section 2 remains unchanged but now only the variables occuring in annotated positions are considered actions of the scheme. To give the sufficient condition mentioned above we prove the following general theorem.

Considering an attributed scheme, we denote by need (ν) the set of all positions from which ν depends. More formally, in a rule r_c :

$$\underline{need}(\nu) = \{\mu \mid \exists a \in A_{r_c}, \mu \in \underline{in}(a) \wedge \nu \in \underline{out}(a)\}$$

THEOREM 1 : Given a program P, a partial-d-assignment and the associated attributed scheme $(G, d, A, \underline{in}, \underline{out})$.

If 1) in each production r_c :

$$\forall \nu \in \text{output}(r_c) : (\forall \mu \in \underline{need}(\nu), \mu \textit{ ground}) \Rightarrow \nu \textit{ ground}$$

2) the attributed scheme is well-formed.

Then for every complete tree τ *with root* X

$$\forall \nu \in \uparrow X, [\forall \mu, \mu \in \underline{use}(\tau,\nu) \wedge \mu \textit{ ground}] \Rightarrow \nu \textit{ ground}$$

PROOF

The theorem is proved by induction, following the model given in [De 83] for Logical Attribute Grammars (here we deal with a particular case).

We show by induction that all positions in all proof trees of P are ground.

If the attribute scheme is well-formed, then the relation $\underset{\tau}{\rightarrow}$ is a partial order. The induction will follows the directed acyclic graph (DAG) corresponding to this partial order. By construction of the attribute schemes, all its minimal elements correspond to local actions without input positions, or to input positions of the root.

By condition 1, left member of conclusion implication, and definition of the partial d-assignment, these positions are ground.

Now consider any other position in the DAG. By the construction it is an output position of some instance of some clause.

By induction hypothesis all the input positions of the actions having this position in their output positions are ground. So by condition 1, it is ground.

Now the question is, which hypothesis have been used to prove that some position of X is ground. Clearly the answer is (by definition of use) : use (τ, ν), the usefull inherited attributes of ν, assuming that all its elements are ground. QED.

By this theorem we know that to obtain a ground synthesized position of some litteral, for a given proof-tree, it is sufficient to make ground all positions of the corresponding useful set. If we consider all the possible proof-trees, we can deduce that if all positions corresponding to the union of all possible useful sets of this position are ground, then the corresponding synthesized position is ground.

Let us denote $\underline{use}(X, \nu)$ the union of all possible usefull sets for some position ν of $\uparrow X$, and $\underline{use}(X)$ the union of all argument selectors of all elements of $\uparrow X$, i.e.:

$$\underline{use}\,(X,\nu) = \bigsqcup_{\tau^X} \underline{use}\,(\tau^X,\nu)$$

$$\underline{use}\,(X) = \bigsqcup_{\nu \in X} \underline{use}(X,\nu)$$

It is clear that for any call, if $\underline{use}(X) \subseteq \downarrow X$ are ground before some evaluation, $\uparrow X$ will be ground after the evaluation of the litteral X.

Now we would like to have some criteria to test the condition-1 of Theorem 1.

THEOREM 2 : Given the attribute scheme associated with P, *then both statements are equivalent :*

(i) in each production r_c *:*

$\forall \nu \in$ output (r_c) : $\{\forall \mu \in$ in $(\nu), \mu$ *ground*$\} \Rightarrow \nu$*ground*

(ii) in each clause, any variable has at least one input position.

PROOF :

The proof is based on construction 1.

(i) $\Rightarrow$ (ii)

Consider an output position ν with a free variable x. By construction 1, this variable is associated an action a such that $x \in \underline{out}$ (a). By hypothesis (i), $\underline{in}$ (ν) ground implies ν ground. So x appears in some input position μ belonging to $\underline{in}$ (ν). By construction 1, $\mu \in \underline{in}$(a). Thus $\underline{in}$ (a) is not empty.

(ii) $\Rightarrow$(i) Obvious, since any variable in the output positions, by construction of the actions occurs in some ground input position.

QED

As a corollary from Theorem 1 and Theorem 2 we obtain at once.

THEOREM 3

If P *is a logic program and* d *is a direction assignment on* P *such that*
(i) the attributed scheme of P *obtained for* d *by Construction 1 is well-formed, and*
(ii) in each clause of P *any variable occuring in the clause has at least one input position*
then d *is correct for* P .

The question arises whether for a given logic program one can find an assignment satisfying the conditions of Theorem 3. Very often such an assignment can be easily deduced, espacially for small programs. A typical situation when no such assignment exists concerns the case when a procedure of a logic program is called in the program more than once with different purposes. For example, consider the following sorting program originating from [CM81].

```
sort(↓L,↑S) <=  append(X,[A,B | Y],L)
                B < A
                append(X,[B,A | Y],M)
                sort(↓M,↑S).

sort(↓L,↑L) ⇐
append([ ],L,L) ⇐
append([E | L1],L2,[E | L3]) ⇐ append(L1,L2,L3).
```

In the first clause the append procedure is called twice. The first call splits the input list into two sublists, the other appends two sublists into a list. It is easy to see that there is no assignment satisfying the conditions of Theorem 3. Since we require that the variables X and M have at leastone input position in the first clause we must assume that the first and the third positions of append are synthesised. But in this case variable E in the last clause of the program has no input position.

Suppose now we accept for append to use different d-assignments, considering we have two programs append1 and append2 with assignments ↑↑↓ and ↓↓↑ By authorizing multi-d-assignment, there is only one possible way to achieve the construction of Theorem 1 with the restriction (ii) of theorem 3.

sort(↓L,↑S) append 1(↑X, ↑[A,B | Y],↓L)
↓B < ↓A
append2 (↓X, ↓[B,A | Y], ↑M)
sort(↓M, ↑S).
sort(↓L,↑L)

Of course append1 and append2 have the same axioms with only different d-assignments which respect the restriction. The resulting scheme gives a clear idea about the way the resolution is processed : append1 splits the list L append2 builds a modified list M.

Moreover, we know with these d-assignments that append programs will terminate.(*) So the whole program will terminate, if the resolution follows the order of choosing the litteral suggested by the dependencies as builded in the section (left to right order).

Remark that the process of duplicating the sets of clauses is finite because there is only a finite number of possible d-assignments for each predicate. But not all programs may have such multi-d-assignment as shown by the following example.

Example 2 : P(X) ⇐ Q(Y) . (1)
Q(X) ⇐ P(Y) . (2)

which can be considered as a part of a bigger HCP.

In fact the only possible d-assignment for the first clause is :

P(↓X) ⇐ Q(↑Y).

(which respects condition (ii) of theorem 3, even if P is defined by using more clauses). But this assignment does not respect the condition in the second clause.

(*) Because inherited positions are suposed to be instanciated with finite ground terms whose size is decreasing through the recursive calls.

5 - Data Driven Evaluation

The left-to-right evaluation strategy used commonly by Prolog interpreters may sometimes have disadvantages. For example, consider the following simple program

```
grandfather(X,Y) <= father (X,Z), father (Z,Y).
father(Peter,Paul) <= .
father(Mary,Georges) <= .
father(Paul,Georges) <= .
....
```

If we consider the goal

```
grandfather(Peter,Y).
```

the best way to solve this goal is to solve the litteral of the clause defining GRANDFATHER in the order thery are given. So both calls to FATHER will be partially instantiated.

On the contrary, if we consider the goal

```
grandfather(X,Georges).
```

it would be better to try to satisfy the second literal before the first to get the same situation.

We would like to improve the efficiency of such a strategy by computing automatically an evaluation order for a family of goals before the evaluation of a goal from this family starts. More precisely, we assume, that it is known in advance which arguments of the goal literals are ground. These arguments are inherited positions in the goal predicates. We expect, that after evaluation of the goal its other (i.e synthesised) positions will be ground. In other words, we assume that a correct partial d-assignments is given, concerning at least the goal predicates.

If such an assignment is given it issometimes possible to use the following evaluation strategy: a procedure call can be evaluated only when all its input arguments are ground. This will be called data driven evaluation. In this section we give a sufficient condition for an assignment which makes it possible to follow the rule stated above. The condition is even more restrictive : it allows to compute statically the order of evaluation of the literals of each clause. To formulate this condition we use the notion of one-sweep attribute scheme [EF82].

For any nonterminal X of an attribute scheme denote by Θ_X the relation on the positions of X defined as follows:

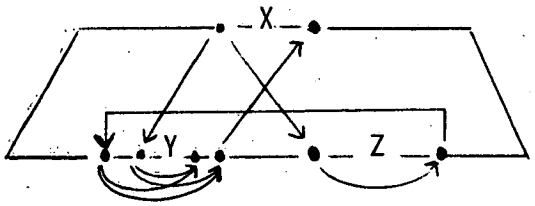
$p \Theta_X q$ iff p is inherited and q is synthesised

DEFINITION 2

An attributed scheme is one-swepp iff in each production: $p : X_0 \leftarrow X_1 \ldots X_n$, *the graph of the relation* $\theta_{X_1} \cup \ldots \theta_{X_n} \cup LD(p)$ *is acyclic*

(LD(p) denotes the local dependencies between input and output positions induced by the actions).

Example 3 :



Example of graph without cycle (one-sweep case)

One the left (right) hand side of a non terminal the positions are ingerited (synthesized).

In this example, Z would be always cadidate for evaluation before Y.

It is possible to use this approach in two ways :

- to preserve the original program and to choose the literals in a fixed order.
- to reorder the literals of the body in the clauses in such a way that a fixed left to right order can be used.

It is worth noticing that the one-sweep category is a subcategory of the well-formed attribute schemes and thus one-sweep schemes are also well-formed.

Nevertheless, this category of attribute schemes is considerably restrictive. It seems that other strategy could be introduced by using fixed ordering on the positions. This would authorize to evaluate partially a predicate. To explore the advantages or drawbacks of this approach is out of the scope of this paper.

Comming back to the example of GRANDFATHER, we could consider three families of goals corresponding to the following d-assignment :

↓↑ = asking if two persons are separated by two generations.

↑↓ = looking for the grandfathers

↓↓ = looking for the grandsons.

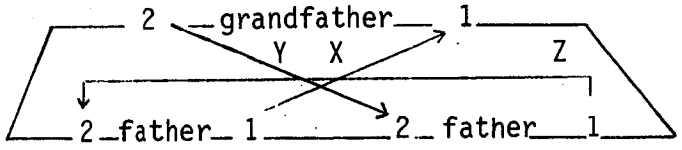
It is also clear that the results will be more efficiently computed if some arguments of FATHER are ground, i.e. with the following possible d-assignments :

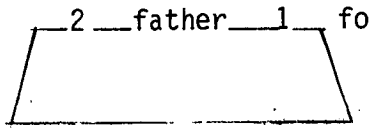
↓↓ : all parameters instantiated

↑↓ } one parameter instantiated
↓↑ }

Note that the d-assignments of GRANDFATHER are given, but the d-assignments of FATHER are "suggestions". Suppose we consider a goal family with ↓↑. It is easy to see that with FATHER's d-assignment ↑↓ satisfying the hypothesis of Theorem 1.

Example 4





for all clauses of FATHER

(no dependency : all positions ground)

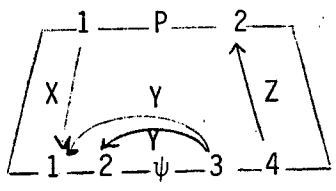
The obtained attribute scheme is trivially non-circular.

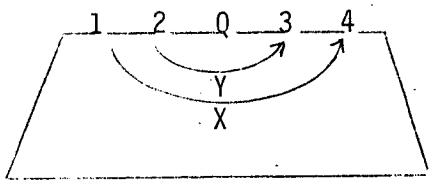
So by Theorem we get a first result showing this assignment is "correct" in the sense that any answer substitution will be ground.

Notice that this approach can be generalized by using multi-d-assignment. So to obtain for this example a d-assignment satisfying conditions of Theorem 2 with the goal family ↓↓, it would be necessary to use two simultaneous d-assignments of FATHER : ↓↑ and ↓↓ In this sense, the Example 4 is also "correct" (the associated attribute scheme is non-circular), but the following is not correct.

Example 5 : P(↓X; ↑Z) ⇐ Q(↓XY,↓Y,↑Y,↑Z) (only variables of the terms are represented here)
Q(X,Y,Y,Z) <= .

The local dependency relations are the following :





Because of the circularity, the fact that X is ground does not imply that Z is ground in the goal P(X,Z).

- Going back to the former example, it is easy to verify that the corresponding attributed scheme is one-sweep and that in the GRANDFATHER clause, the second instance of FATHER has always (for the goal assignment ↑↓) to be choosen before the first one.

The class of logic programs which have a corresponding one-sweep schemes seems to be rather restricted. Nevertheless, many of the examples published in the literature falls in this class. Moreover, one can model an arbitrary Turing machine by an annotated logic program, whose associated attribute scheme is one-sweep. The construction goes as follows.

Any instantaneous description of a given Turing machine T is modelled by a ground term of the form id(l ,r,q), where l is a list describing the tape of T to the left of the head in the reverse order, l is a list describing the tape of T to the right of the head, including the scanned symbol, and q is the actual state of T. Thus, the instantaneous description abqcd where a,b,c,d are tape symbols and q is a state will be represented by the term id ([b|a],[c|d], q).

A move of T in the state q for a scanned symbol x is determined by the transition function ; q is replaced by some q', X by some symbol X' and the head may be moved, for example to the left. Since the number of such elementary moves is finite they can be described by a finite database of facts. For example, the move described above can be represented as follows :

move (id[Z|Y], [x|X],q), id[Y,[Z,x'|X],q')) ⇐ .

The machine, when started in some initial intantaneous description I continues until it reaches a final state q_f (it may also interrupt the computation if the next move is undefined). The result of a successful computations is the final instantaneous description. Thus :

machine (I,I) ⇐ final(I).

machine (I,F) ⇐ move(I,X), machine (X,F).

final (id(Y,X,q_f)) ⇐ .

6 - Conclusion

We have shown that some methods developed for attribute grammars can be applied to study properties of logic programs. In particular we considered the possibility of modelling data flow during the computation of a logic program by means of attribute schemes. As a result we got two restricted but non-trivial classes of logic programs :

- the programs whose data flow can be modelled by attribute schemes : this is the class of programs who have some correct direction assignments ;
- the programs, whose evaluation may be data-driven in a restricted sense defined in this paper.

A logic programming environnement could include tests for checking whether a given program is in one the classes mentioned above. It remains to comment on complexity of such tests.

This problem can be splitted into two subproblems :

- Given a HCP, compute all (multi) d-assignments satisfying some condition on the variables.
- Test the category of the obtained attribute scheme.

The first one is similar to that formulated in [FH79] for extended attribute grammars. In the worst case it may have the full combinatorial complexity given by the number of possible d-assignments for a given HCP. Of course conditions as exhibited in Theorem 3 or even more restrictive, as suggested in section 3, will drastically limit the number of attribute schemes, in such a way that we can expect to get them with a sufficient practical efficiency.

Although such conditions will restrict the class of logic programs to be concerned by this approach, we claim that a large number of realistic and well structured logic programs will fall down in this category.

The second problem corresponds to the test of attribute scheme categories. In this paper we have considered only two categories : non circular and one-sweep.

To test if some attribute grammar belongs to some category takes exponential time for the first one and polynomial time for the second, depending on the size of the attribute scheme, i.e., the size of the HCP. All these results are synthetized in the Figure 5.

Figure 5

Inclusion of categories and test complexity of attribute schemes.

It is also well-known [DJL83] that in meta-compilation optimization on the exponential test can be introduced in such a way that even in big grammars the combinatorial explosion does not appear. It remains to verify wether these optimizations will produce the same effect into HCP. In any case it is worth to consider such a test because it will be performed at "compiletime" and not at execution time. Experimentation on the complexity of the presented constructions is currently processed at INRIA.

From pragmatic point of view the notion of data driven evaluation as introduced in this paper seems to be too restrictive. We hope that our definitions can be refined. In particular, in order to compare our work with the approach presented in [BLM83] or in [NI83] we should have assign directions to variables occuring in the clauses of a program rather than to the positions of the predicate. On the other hand, the Turing machine example shows that, at least from theoretical point of view, our restrictions are not strong limitations.

We hope that the approach presented in this paper has other interesting applications. Generally speaking it gives a method for proving properties of logic programs, and makes it possible to define some classes of logic programs. In particular one may try to use it to formulate sufficient conditions for termination and sufficient conditions for making occur-check unnecessary [DM84]. One can also consider the possibility of replacing resolution by an attribute evaluation process, at least for a class of logic programs. This may be particularly interesting in case of Definite Clause Grammars since the strings of the language can be used to facilitate construction of parse trees of the associated attribute schemes.

REFERENCES

[BLM83] M. BELLIA, G. LEVI, M. MARTELLI
On Compiling Prolog Programs on Demand Driven Architectures.
Proceedings on Logic Programming Workshop 83, Portugal, June 1983.

[CM81] W.F. CLOCKSIN, C.S. MELLISH
Programming in Prolog.
Springer Verlag, Berlin, 1981.

[De83] P. DERANSART
Logical Attribute Grammars.
IFIP 83, R.E.A. Mason (ed), Elsevier Science Publishers B.V.
(North Holland), pp. 463-469.

[DJL83] P. DERANSART, M. JOURDAN, B. LORHO
Speeding up Circularity tests for Attribute Grammars.
R.R. n° 211, INRIA, May 1983, to appear in Acta Informatica.

[EF82] J. ENGELFRIET, G. FILE
Passes, sweeps and visits in Attribut Grammars.
Twente University of Technology, Enschede, The Netherlands, Memorandum
INF 82-6.

[FH79] H.F. FRANZEN, B. HOFFMANN
Automatic Determination of Data Flow in Extended Affix Grammars.
TU Berlin, Project EAGLE, 9th Annual GI Conference, Bonn, 1979.

[Mel81] C.S. MELLISH
Automatic Generation of Mode Declarations for Prolog Programs.
DAI, Edinburgh, Draft, 1981.

[Nil83] J.F. NILSSON
Data Flow Schemes for Prolog.
DCS, Technical University of Denmark, Draft, 1983.

[PW80] F.C.N. PEREIRA, D.H.D. WARREN
Definite Clause Grammars for Language Analysis. A Survey of the Formalism and a Comparison with Argumented Transition Networks. Artificial Intelligence 13, 1980, pp. 231-278.

[Rä80] K.J. RÄIHÄ
Bibliography on Attribute Grammars.
SIGPLAN Notices 15, 3, (1980), 35-44.

[DM84] P.DERANSART, J. MALUSZYNSKI.
Relating Logic Programs and Attribute Grammars. INRIA and Linköping report. 1984 - (To appear).

[Kow74] R. KOWALSKI.
Predicate Logic as Programming Language. IFIP74. Noth-Holland Publishing Company (1974) 569-574.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique